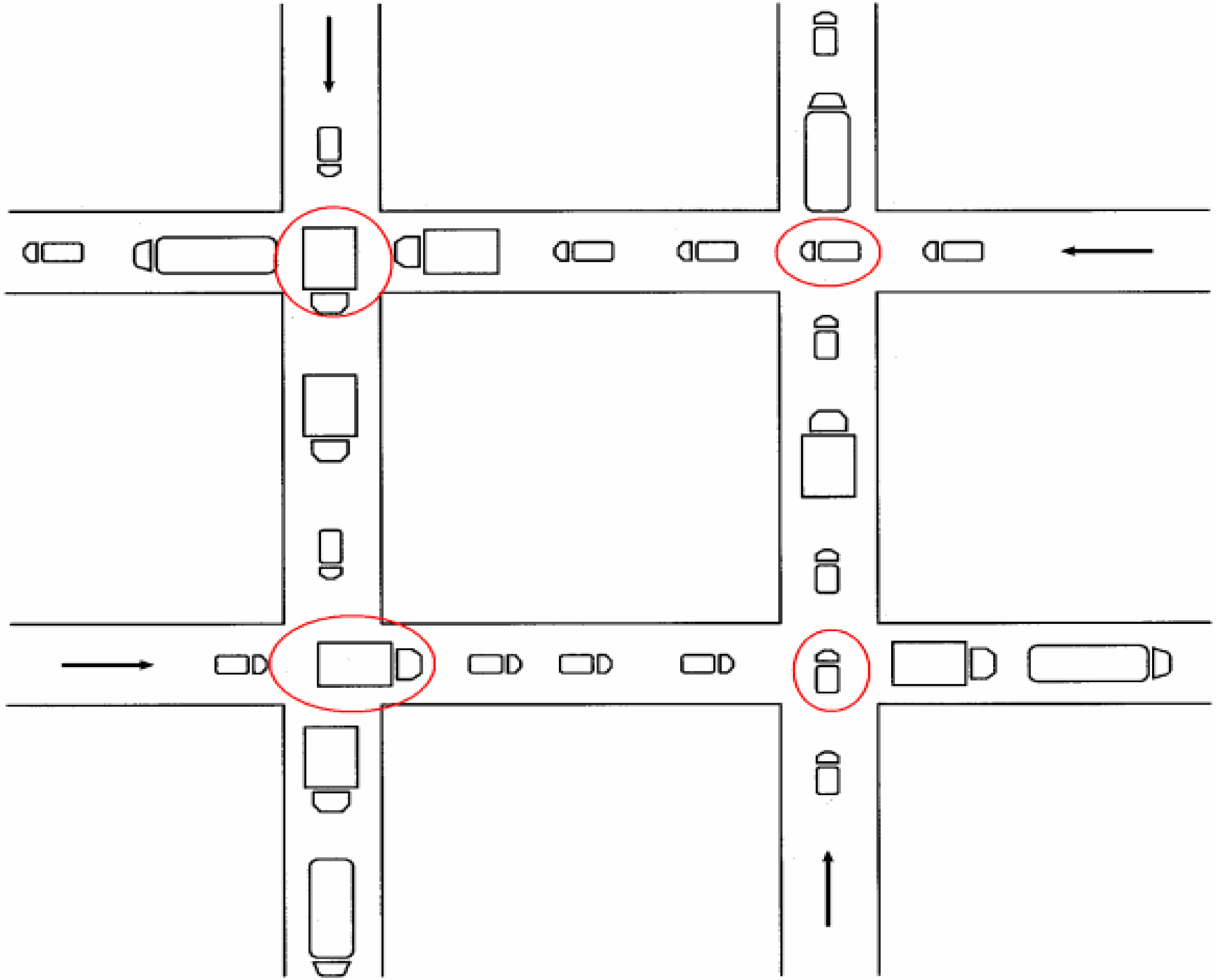
Deadlock

By

Lecturer: Ameen A.Noor

Deadlock

- A Computer System consist of a finite number of resources to be distributed among a number of computing processes.
- The resources are partitioned into several types, each of which consists of some number of identical instances memory, CPU cycles, files, and I/O devices are examples of resource types.



Deadlock

- Under the normal of operation, a process may utilize a resource in only the following sequence:

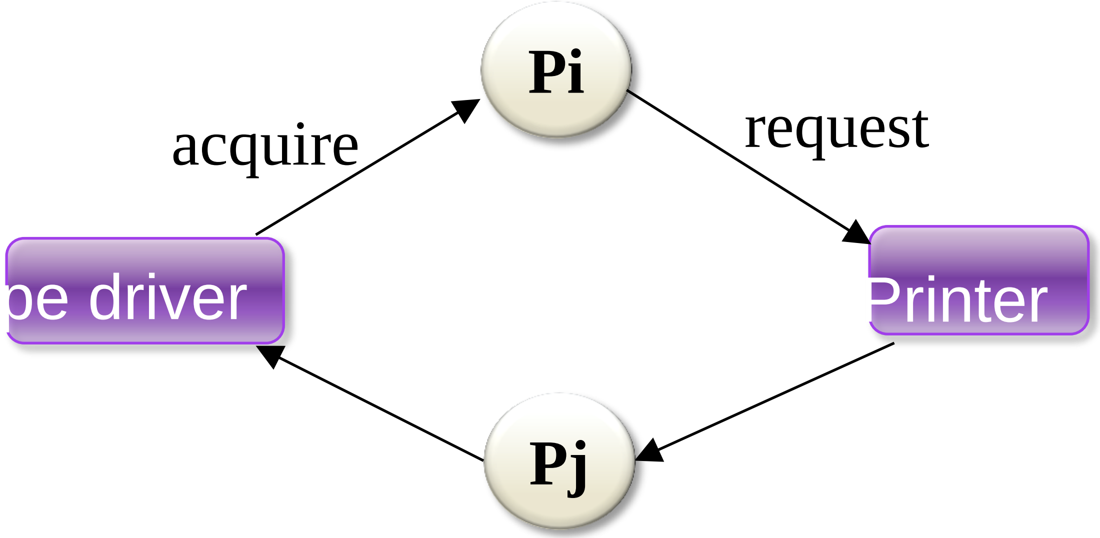
  a. **Request:** If the request cannot be granted immediately then the requesting process must wait until it can acquire the resource.

  b. **Use:** The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).

  c. **Release:** The process releases the resource.

Deadlock definition

- A set of processes each holding a resource and waiting to acquire a resource held by another process in the set.



Deadlock Necessary Conditions

a. **Mutual Exclusion:** Only one process can use a particular resource at the specified time.

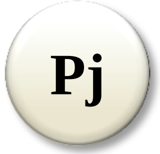
b. **Hold and Wait:** The process maintains at least one resource, and is waiting for additional resources currently being held by other operations.

c. **No Preemption:** Resources cannot be preempted, that is a resource can be released only voluntarily by the process holding after that process has completed its task.

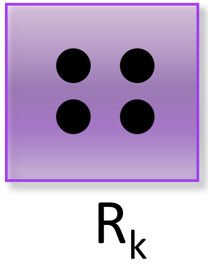
d. **Circular Wait:** There must exist a set $\{p_0,p_1,\ldots.,p_n\}$of waiting processes such that p_0 is waiting for resource that is held by p_1, p_1 is waiting for a resource that is held by p_2, …p_{n-1} is waiting for a resource that is held by p_n, and p_n is waiting for a resource that is held by p_0.

Resource Allocation Graph

- A set of processes {P_0, P_1, …}



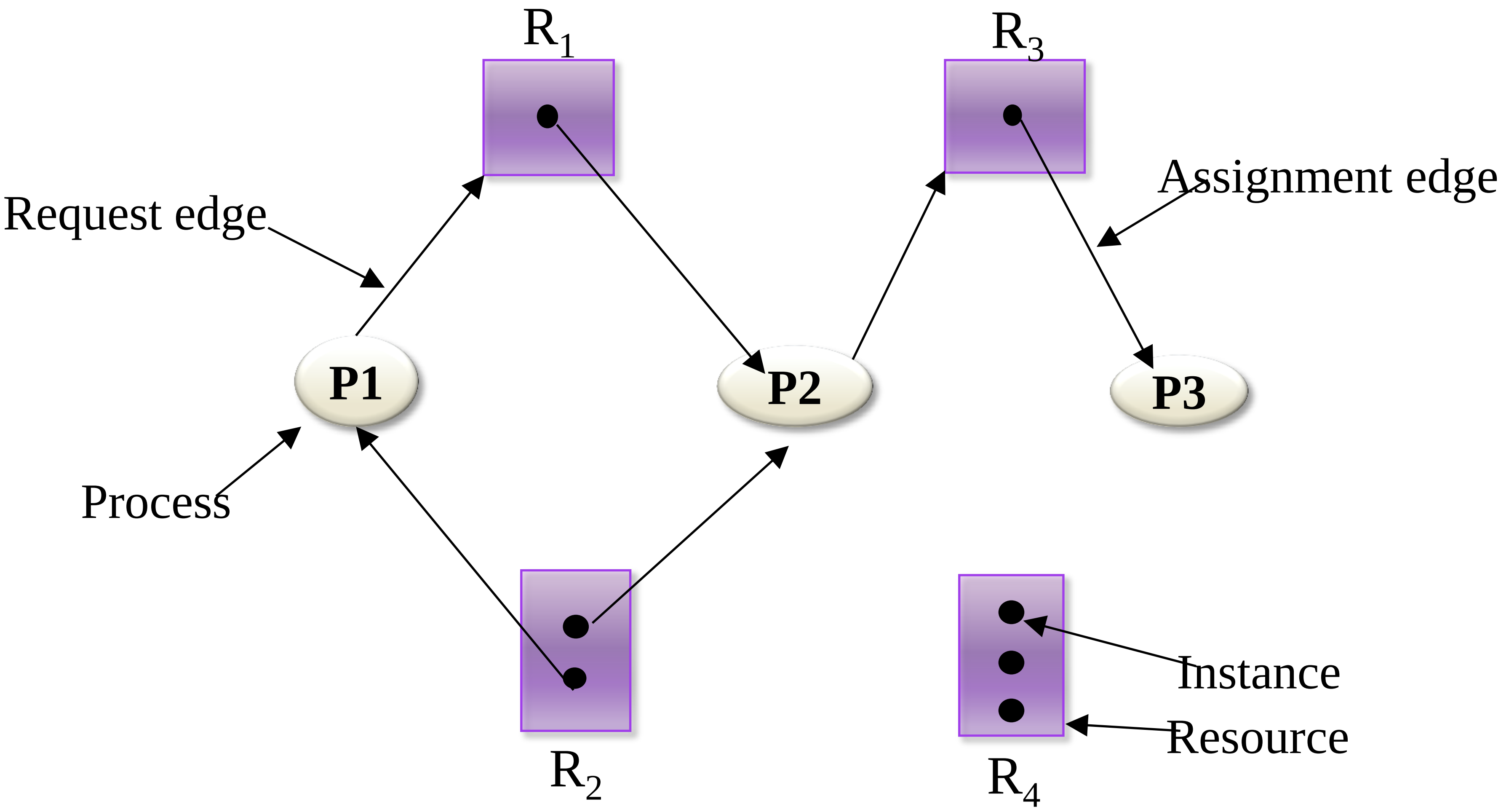
- A set of resource types {R_1, R_2, …}, together with instances of those types



Edge Notation

- $P_i \rightarrow R_j$
  - process i has requested an instance of resource j
  - called a ***request edge***
- $R_j \rightarrow P_i$
  - an instance of resource j has been assigned to process i
  - called an ***assignment edge***

Example Graph



Process states:

P1 → R1, P2 → R3, R1 → P2, R2 → P1, R2 → P2, R3 → P3

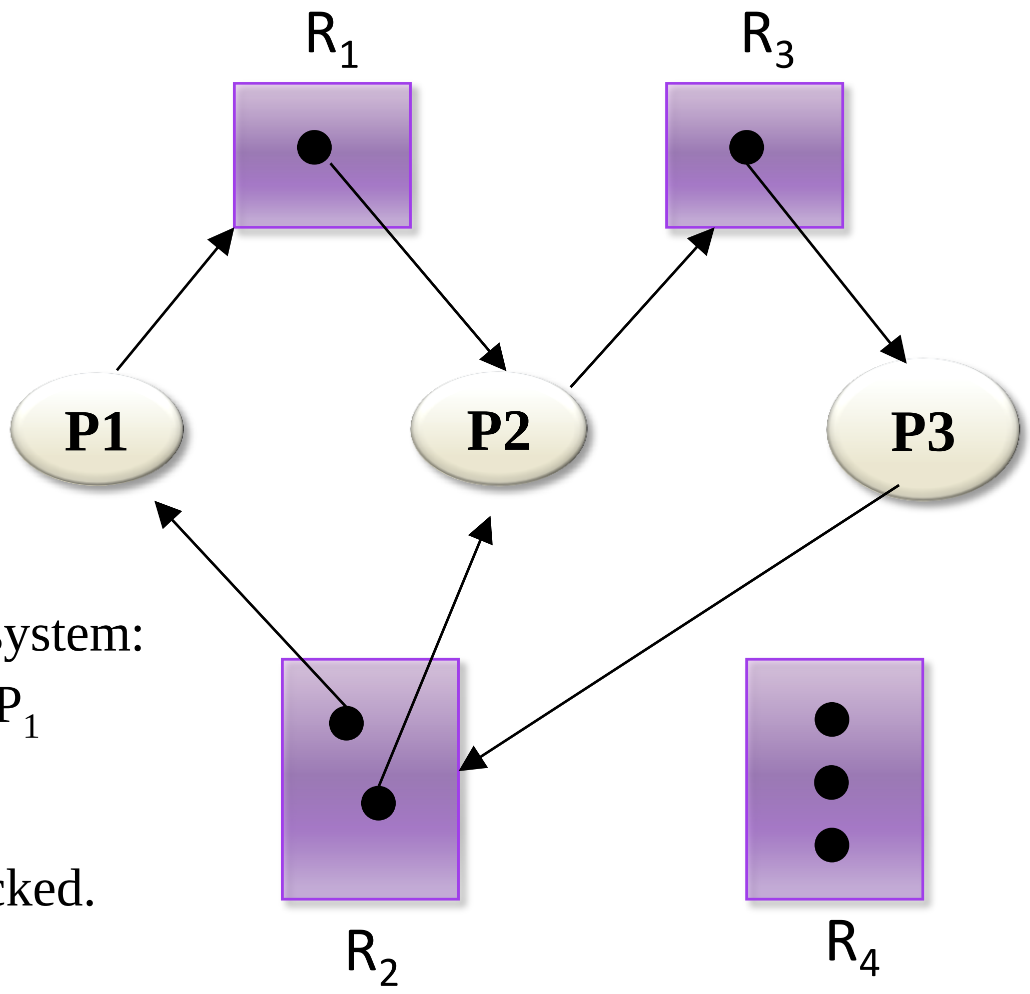
Resource Allocation Graph

- By using a RAG it can be easily shown that if the graph contain **no cycles, then no process in the system is deadlocked**.
- On the other hand if the graph **contains a cycle** then a **deadlock may exist**.
- If the cycle involves only a set of resource types each of which has only a single instance then a **deadlock has occurred.**
- In this case a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.

Resource Allocation Graph

- If each resource type has several instances then a cycle does not necessarily imply that a deadlock occurred.
- In this case a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

Graph with Deadlock



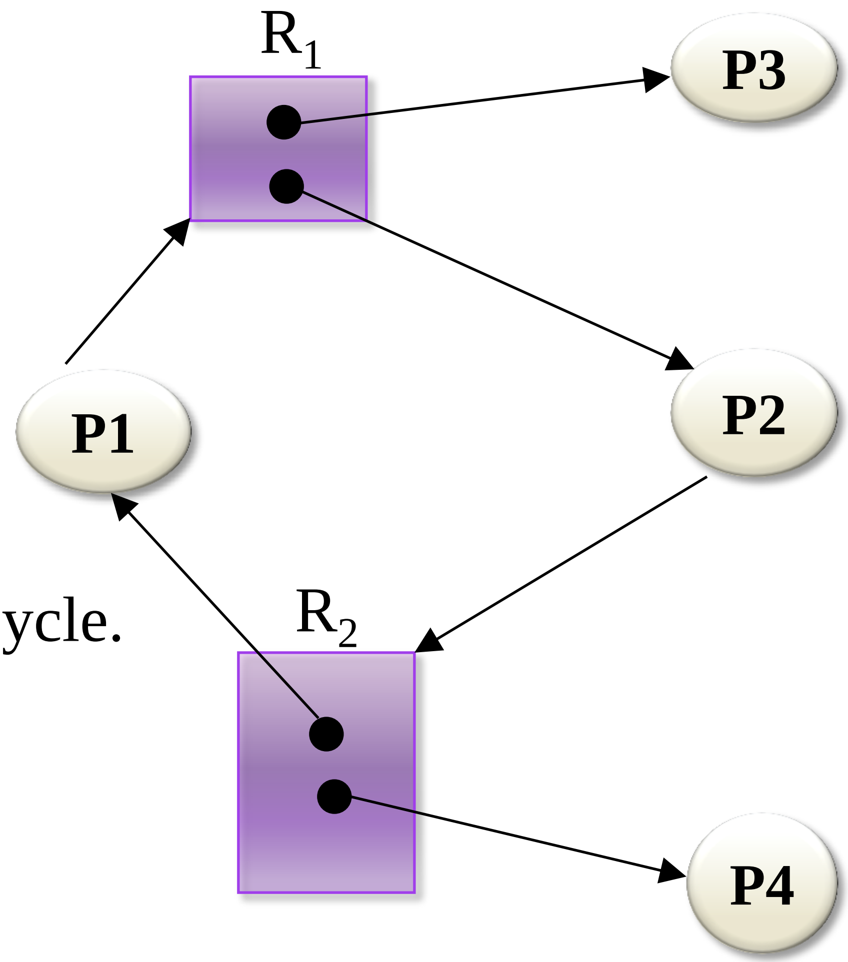
Two minimal cycles exist in the system:

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

Process P_1, P_2 and P_3 are deadlocked.

Graph without Deadlock



In this example, we also have a cycle.

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_2 \rightarrow P_1$

However, there is **no deadlock.**

Methods for Handling Deadlock

a. We can use a protocol to ensure the system will never enter a deadlock state.
b. Allow the system to enter a deadlock state and then recover.
c. We can ignore the problem all together and pretend that deadlocks never occur in the system.

Deadling with Deadlocks

- Stop a deadlock ever occurring
  - **deadlock prevention**
    - disallow at least one of the necessary conditions
  - **deadlock avoidance**
    - Does not meet the request if the process was causing deadlock

Deadlock Prevention

a. **Mutual-exclusion:** The mutual-exclusion condition must hold for non–sharable resource. For example, a printer cannot be simultaneously shared by several processes. Sharable resources on the other hand do not require mutually exclusive access, and thus cannot be involved in a deadlock.

b. **Hold and Wait:** To ensure that hold-and-wait condition never occurs in the system must guarantee that whenever a process request a resource it does not hold any other resources.

c. **No – preemption:** If a process that is holding some resources request another resources that cannot allocated to it then all resources currently being held are preempted.

Deadlock Prevention

d. **Circular wait**: Let R = { R_1 , R_2 , R_3 , ….. , R_n } be the set of resource types. We can assign to each type a unique integer number which allow us to compare two resources.

F(tape drive)=1

F(disk drive)=5

F(printer)=12

We can now consider the following protocol to prevent deadlocks: Each process can request resources only in an increasing order of enumeration. That is process initially request any number of instances of R_i after that the process can request instances of resource type R_j if and only if $F(R_j) > F(R_i)$.

Deadlock Avoidance

- Prevent deadlocks by restraining how requests can be made.
- The restraints ensure that at least one of the necessary conditions for deadlock cannot occur, and, hence, that deadlocks cannot hold.
- Possible side effects of preventing deadlocks by this method, however, are low device utilization and reduced system throughput.

Safe State

- A state is safe if the system can allocate resources to each process (up to maximum) in some order and still avoid a deadlock.
- A safe state is not a deadlock state, and a deadlock state is an unsafe state, but not all unsafe states are deadlock. An unsafe state may lead to a deadlock.

Example: to illustrate consider a system with 12 magnetic tape drives
3 process (P0 , P1 , P2).

Process P0 requires 10 tape drives, process P1 may need as many as 4, and process P2 may need up to 9 tape drives.

Suppose that, at time t0, process P0 is holding 5 tape drives, process P1 is holding 2, and process P2 is holding 2 tape drives. (Thus, there are 3 free tape drives). The maximum needs and current needs for each process as indicated below:

	Maximum needs	**Current needs**	**Available**
P_0	10	5	3
P_1	4	2	
P_2	9	2	

At time t_0, the system is in a safe state. The sequence $< P_1 , P_0 , P_2 >$ satisfies **the safety condition.**

deadlock avoidance

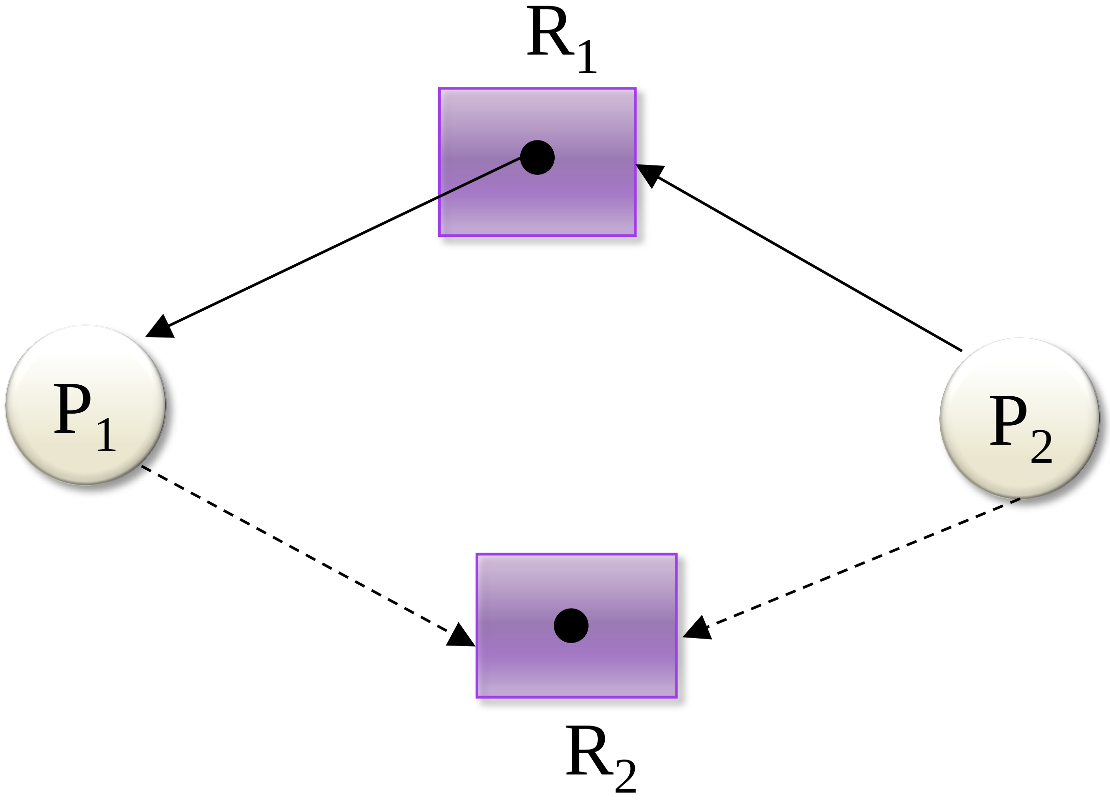
There are many deadlock avoidance algorithms, some of these are:

- **Resource-Allocation Graph Algorithm**
  - If we have a RAG system with only one instance of each resource type. In addition to the request and assignment edges we introduce a new type of edge called a **claim edge**.
  - A claim edge Pi → Rj indicates that process P_i may request resource R_j at some time in the future.
  - This edge as a request edge in direction but is represented by a dashed- line.

Resource-Allocation Graph Algorithm

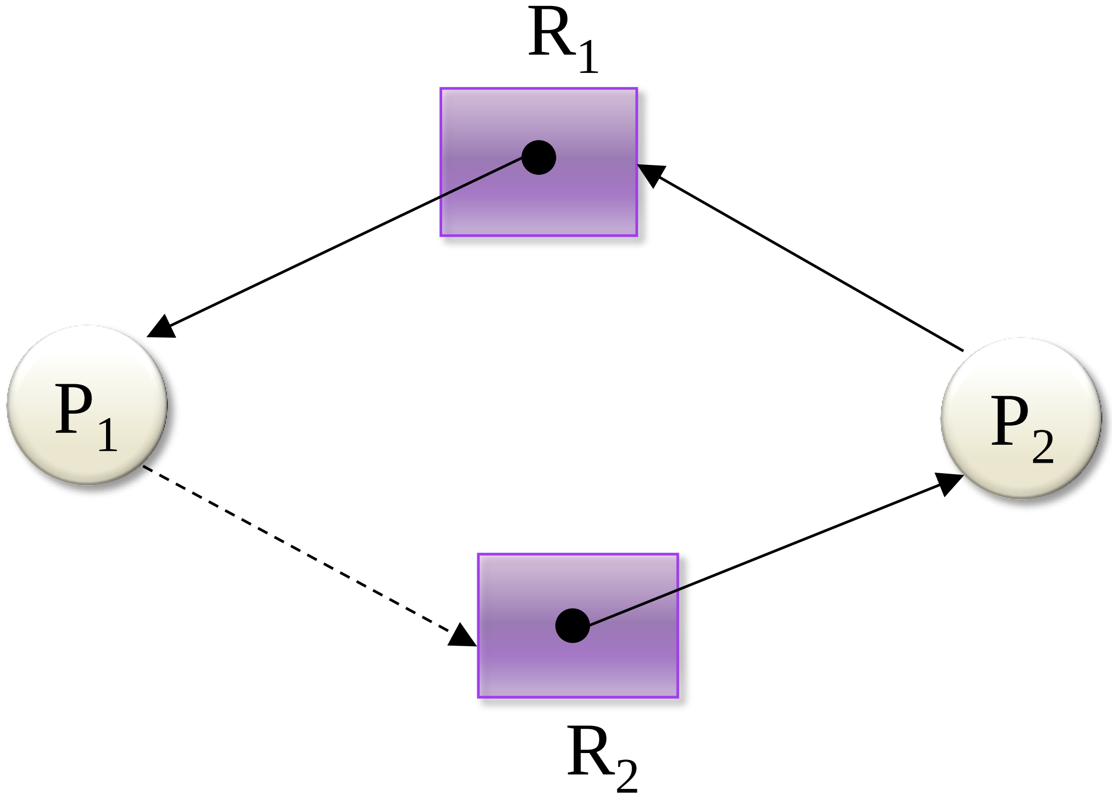
- when process P_i request R_j the claim edge Pi → Rj is converted to a request edge.
- When a resource R_j is released by P_i the assignment edge Rj → Pi is reconverted to a claim edge Pi → Rj .

Resource-Allocation Graph Algorithm



RAG for deadlock avoidance

Resource-Allocation Graph Algorithm



Unsafe state in RAG

Banker’s Algorithm

- The banker’s algorithm which is also known as avoidance algorithm is a deadlock detection algorithm.
- It is designed to check the safe state whenever a resource is requested.
- When a new a process enters the system it must declare the maximum number of instances of each resource type that it may need.
- The maximum must be ≤ total number of resources in the system.
- When a user requests a set of resources must be leave the system in a safe state.

Banker's Algorithm

- Several data structures must be maintained to implement banker's algorithm. We need the following data structures:
  - **Available:** indicates the number of available resources of each type. If available[j]=k these are k instances of resource type R_j available.
  - **Max:** defines the maximum demand of each process. If max[i,j] = k then process P_i may request at most k instances of resource type R_j
  - **Allocation:** the resources currently allocated to each process. If allocation[i,j] = k then process P_i is currently allocated k instances of resource type R_j.
  - **Need:** the remaining resource need of each process. If Need[i,j]=k then process P_i may need k more instances of resource type R_j to complete its task.

    Need[i,j] = Max [i,j] – Allocation [i,j]

Example of Banker's Algorithm

- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances), *B* (5instances), and *C* (7 instances).

- Snapshot at time T_0:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example (Cont.)

- The content of the matrix *Need* is defined to be *Max* – *Allocation*.

Need

	A	*B*	*C*
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria.

Example of Safe State:

Suppose a system contains 12 resources and three processes sharing the resources, as in table below.

	Max	Allocated	Current need	Available
P_1	4	1	3	2
P_2	6	4	2	
P_3	8	5	3	

The sequence $< P_2 , P_1 , P_3 >$ satisfies **the safety condition.**

Example of Unsafe State:

Suppose a system contains 12 resources and three processes sharing the resources, as in table below.

	Max	Allocated	Current need	Available
P_1	10	8	2	1
P_2	5	2	3	
P_3	3	1	2	

Deadlock Detection

- Deadlock detection is the process of determining that a deadlock exists and identifying the processes and resources involved in the deadlock.
- Deadlock detection algorithms generally focus on determining if a circular wait exists, given that the other necessary conditions for deadlock are in place.
- In this environment, the system must provide:
    - An algorithm that examines the state of the system to determine whether a deadlock has occurred.
    - An algorithm to recover from the deadlock.

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances).
- Snapshot at time T_0:

	Allocation			*Request*			*Available*		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish*[*i*] = true for all *i*.

Example (Cont.)

- P_2 requests an additional instance of type *C*.

	Request		
	A	*B*	*C*
P_0	0	0	0
P_1	2	0	1
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests.
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.

Recovery from deadlock

- When a detection Algorithm determines that a deadlock exists the system must recover from the deadlock.
- There are two options for breaking a deadlock

a. **Process termination** by killing a process, two methods:

  - Kill all deadlocked processes.
  - Kill one process at a time until the deadlock cycle is eliminated.

b. **Resource preemption:** to eliminate deadlocks using resource preemption we can preempt some resources from processes and give them to other processes until the deadlock cycle is broken.

Recovery from deadlock

- If preemption is required in order to deal with deadlocks then three issues need to be addressed:
  - **Selecting a victim:** which process and which resources.
  - **Rollback:** if we preempt a resource from a process what should be done with that process?
  - **Starvation.**